LAB 7

IT 314 Software Engineering

Krish Gulabani 202001053

Section A:

Tester Action and Input Data	Expected Outcome
Equivalence Partitioning	
Invalid Month (< 1)	Error
Invalid Month (> 12)	Error
Valid Month, Invalid Day (< 1)	Error
Valid Month, Invalid Day (> 31)	Error
Valid Month, Valid Day, Invalid Year (< 1900)	Error
Valid Month, Valid Day, Invalid Year (> 2015)	Error
Valid Month, Valid Day, Valid Year	Previous date or Invalid date

Boundary Value Analysis	
Month is 1, Day is 1, Year is 1900	Error
Month is 12, Day is 31, Year is 2015	Previous date or Invalid date
Month is 2, Day is 29, Year is 1900	Error
Month is 2, Day is 28, Year is 1900	Previous date or Invalid date
Month is 2, Day is 29, Year is 2000	Previous date or Invalid date
Month is 2, Day is 28, Year is 2000	Previous date or Invalid date
Month is 4, Day is 31, Year is 2010	Error
Month is 4, Day is 30, Year is 2010	Previous date or Invalid date

P1

The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

```
int linearSearch(int v, int a[])
{
      int i = 0;
      while (i < a.length)
      {
            if (a[i] == v)
                  return(i);
            i++;
      }
      return (-1);
}
```

Equivalence Partitioning:

Tester Action and Input Data	**Expected Outcome**
Input with a valid value that exists in the array v = 7, a = [2, 4, 7, 9, 11]	2
Input with a valid value that does not exist in the array v = 6, a = [3, 8, 12, 15]	-1
Input with a valid value at the first index of the array v = 10, a = [10, 20, 30, 40]	0

Boundary Value Analysis:

Tester Action and Input Data	**Expected Outcome**
Input with the minimum valid array length v :5, a : [ ]	-1
Input with the maximum valid array length v : 25, a : [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]	3
Input with the value not found in a large array v: 200 a: [50, 100, 150, 250, 300, 350, 400, 450, 500]	-1
Input with the value found at the last index of the array v: 60 a: [30, 40, 50, 60]	3
Input with the value found at the first index of a large array v: 5 a: [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]	0

Modified Code:

```
public class tinearSearch {

    public static int linearSearch(int v, int[] a) { int i =
        0;
        while (i < a.length) { if
            (a[i] == v)
                return i;
            i++;
        }
        return -1;
    }

    public static void main(String[] args) {
        System.out.println("Test cases for Equivalence Partitioning: "); int v1 =
        7;
        int[] a1 = {2, 4, 7, 9, 11};
        System.out.println("\nTest Case 1: " + linearSearch(v1, a1)); // Expected : 2

        int v2 = 6;
        int[] a2 = {3, 8, 12, 15};
        System.out.println("\nTest Case 2: " + linearSearch(v2, a2)); // Expected : -1

        int v3 = 10;
        int[] a3 = {10, 20, 30, 40};
        System.out.println("\nTest Case 3: " + linearSearch(v3, a3)); // Expected : 0

        System.out.println("\nTest cases for Boundary Value Analysis: ");


        int v4 = 5; int[]
        a4 = {};
        System.out.println("\nTest Case 4: " + linearSearch(v4, a4)); // Expected : -1

        int v5 = 25;
        int[] a5 = {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100};
        System.out.println("\nTest Case 5: " + linearSearch(v5, a5)); // Expected : 3

        int v6 = 200;
        int[] a6 = {50, 100, 150, 250, 300, 350, 400, 450, 500};
        System.out.println("\nTest Case 6: " + linearSearch(v6, a6)); // Expected : -1

        int v7 = 60;
        int[] a7 = {30, 40, 50, 60};
        System.out.println("\nTest Case 7: " + linearSearch(v7, a7)); // Expected : 3

        int v8 = 5;
        int[] a8 = {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100};
        System.out.println("\nTest Case 8: " + linearSearch(v8, a8)); // Expected : 0



    }
}
```

Based on modified code outputs are given below:
<u>Output:</u>

```
Test cases for Equivalence Partitioning:

Test Case 1: 2
Test Case 2: -1
Test Case 3: 0

Test cases for Boundary Value Analysis:

Test Case 4: -1
Test Case 5: 3
Test Case 6: -1
Test Case 7: 3
Test Case 8: 0
```

P2

The function countItem returns the number of times a value v appears in an array of integers a.

```
int countItem(int v, int a[])

{
      int count = 0;
      for (int i = 0; i < a.length; i++)
      {
            if (a[i] == v)
                  count++;
      }
      return (count);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v = 7, a = [2, 4, 7, 9, 11]	1
v = 6, a = [3, 8, 12, 15]	0
v = 10, a = [10, 20, 30, 40]	1

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
v = -1, a = [] 0	0
v = 0, a = [0]	1
v = 5, a = [5, 5, 5, 5, 5, 5, 5]	7
v = -999, a = [999]	0

Modified Code:

```
public static int countItem(int v, int[] a) { int count
    = 0;
    for (int i = 0; i < a.length; i++) { if (a[i]
        == v)
            count++;
    }
    return count;
```

Based on modified code outputs are given below:
Output:

```
Test cases for Equivalence Partitioning:

Test Case 1: 1
Test Case 2: 0
Test Case 3: 1

Test cases for Boundary Value Analysis:

Test Case 4: 0
Test Case 5: 1
Test Case 6: 7
Test Case 7: 0
```

P3

The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.

Assumption: the elements in the array are sorted in non-decreasing order.

```
int binarySearch(int v, int a[])
{
      int lo,mid,hi;
      lo = 0;
      hi = a.length-1;
      while (lo <= hi)
      {
            mid = (lo+hi)/2;
            if (v == a[mid])
                  return (mid);
            else if (v < a[mid])
                  hi = mid-1;
            else
                  lo = mid+1;
      }
      return(-1);
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
v = 5, a = [1, 2, 3, 4, 5, 6]	4
v = 9, a = [1, 2, 3, 4, 5, 6]	-1
v = 1, a = [1, 1, 1, 1, 1, 1]	0

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
v = 7, a = [1, 2, 3, 4, 5, 6]	-1
v = 1, a = [1]	0
v = 5, a = [1, 2, 3, 4, 5]	4
v = 1, a = []	-1

Modified Code:

```
public static int binarySearch(int v, int a[]) { int lo,
    mid, hi;
    lo = 0;
    hi = a.length - 1;
    while (lo <= hi) {
        mid = (lo + hi) / 2; if (v
        == a[mid])
            return (mid); else if
        (v < a[mid])
            hi = mid - 1; else
            lo = mid + 1;
    }
    return (-1);
}
```

Based on modified code outputs are given below:
Output:

```
Test cases for Equivalence Partitioning:

Test Case 1: 4
Test Case 2: -1
Test Case 3: 0

Test cases for Boundary Value Analysis:

Test Case 4: -1
Test Case 5: 0
Test Case 6: 4
Test Case 7: -1
```

P4

The following problem has been adapted from The Art of Software Testing, by G. Myers (1979). The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;
int triangle(int a, int b, int c)
{
      if (a >= b+c || b >= a+c || c >= a+b)
           return(INVALID);
      if (a == b && b == c)
           return(EQUILATERAL);
      if (a == b || a == c || b == c)
           return(ISOSCELES);
      return(SCALENE);
```

Equivalence Partitioning

Tester Action and Input Data	Expected Outcome
a = 5, b = 5, c = 5	(Equilateral triangle)
a = 5, b = 5, c = 6	(Isosceles triangle)
a = 3, b = 4, c = 5	(Scalene triangle)
a = 0, b = 0, c = 0	(Invalid triangle)
a = 3, b = 4, c = 8	(Invalid triangle)

Boundary Value Analysis:

Tester Action and Input Data	Expected Outcome
a = 1, b = 1, c = 1	(Equilateral triangle)
a = 1, b = 1, c = 2	(Isosceles triangle)
a = 1, b = 2, c = 1	(Isosceles triangle)
a = 2, b = 1, c = 1	(Isosceles triangle)
a = 1, b = 2, c = 3	(Scalene triangle)
a = 3, b = 4, c = 5	(Scalene triangle)
a = 0, b = 0, c = 0	(Invalid triangle)
a = 1, b = 2, c = 4	(Invalid triangle)
a = 3, b = 4, c = 8	(Invalid triangle)

Modified Code:

```
public int triangle(int a, int b, int c) {
    if (a >= b + c || b >= a + c || c >= a + b) return
        (INVALID);
    if (a == b && b == c) return
        (EQUILATERAL);
    if (a == b || a == c || b == c) return
        (ISOSCEtES);
    return (SCALENE);
}
```

Based on modified code outputs are given below
Output:

```
Test cases for Equivalence Partitioning:

Test Case 1: Equilateral triangle
Test Case 2: Isosceles triangle
Test Case 3: Scalene triangle
Test Case 4: Invalid triangle
Test Case 5: Invalid triangle


Test cases for Boundary Value Analysis:


Test Case 6: Equilateral triangle
Test Case 7: Isosceles triangle
Test Case 8: Isosceles triangle
Test Case 9: Isosceles triangle
Test Case 10: Scalene triangle
Test Case 11: Scalene triangle
Test Case 12: Invalid triangle
Test Case 13: Invalid triangle
Test Case 14: Invalid triangle
```

P5

The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2 (you may assume that neither s1 nor s2 is null).

```
public static boolean prefix(String s1, String s2)
{
      if (s1.length() > s2.length())
      {
            return false;
      }
      for (int i = 0; i < s1.length(); i++)
      {
            if (s1.charAt(i) != s2.charAt(i))
            {
                  return false;
            }
      }
      return true;
}
```

Equivalence Partitioning:

Tester Action and Input Data	Expected Outcome
s1 is a prefix of s2. prefix("hello", "helloworld")	true
s1 is not a prefix of s2. prefix("java", "javascript")	false
s1 and s2 are empty strings. prefix("", "")	true
s1 is null. prefix(null, "world")	NullPointerException
s2 is null. prefix("hello", null)	NullPointerException
s1 and s2 are null. prefix(null, null)	NullPointerException

Boundary Value Analysis:

Tester Action and Input Data	**Expected Outcome**
s1 and s2 are minimum length strings. prefix("a", "a")	true
s1 and s2 are maximum length strings. prefix("abcdefghijklmnopqrstuvwxyz", "abcdefghijklmnopqrstuvwxyz")	true
s1 is an empty string and s2 is a non-empty string. prefix("", "hello")	true
s1 is a non-empty string and s2 is an empty string. prefix("hello", "")	false
s1 is longer than s2. prefix("world", "hi")	false

Modified Code:

```
public static boolean prefix(String s1, String s2) { if
    (s1.length() > s2.length()) {
        return false;
    }
    for (int i = 0; i < s1.length(); i++) { if
        (s1.charAt(i) != s2.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

Based on modified code outputs are given below:
Output:

```
Test cases for Equivalence Partitioning:

Test Case 1: true
Test Case 2: false
Test Case 3: true
Test Case 4: NullPointerException
Test Case 5: NullPointerException
Test Case 6: NullPointerException


Test cases for Boundary Value Analysis:

Test Case 7: true
Test Case 8: true
Test Case 9: true
Test Case 10: false
Test Case 11: false
```

P6

Consider again the triangle classification program (P4) with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled.

Determine the following for the above program:

a) Identify the equivalence classes for the system
b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class.
(Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)
c) For the boundary condition $A + B > C$ case (scalene triangle), identify test cases to verify the boundary.
d) For the boundary condition $A = C$ case (isosceles triangle), identify test cases to verify the boundary.
e) For the boundary condition $A = B = C$ case (equilateral triangle), identify test cases to verify the boundary.
f) For the boundary condition $A^2 + B^2 = C^2$ case (right-angle triangle), identify test cases to verify the boundary.
g) For the non-triangle case, identify test cases to explore the boundary.
h) For non-positive input, identify test points.

a) Equivalence Class

Equivalence Class	Condition
Valid triangle	A + B > C, B + C > A, C + A > B
Scalene triangle	All sides have di erent lengths
Isosceles triangle	Two sides have equal lengths
Equilateral triangle	All three sides have the same length
Right-angle triangle	One angle is equal to 90 degrees
Non-triangle	One of the conditions for a valid triangle is not met

b) Test Cases:

Test Case	Condition	Expected Outcome
1	A=5, B=5, C=5	Equilateral triangle
2	A=3, B=4, C=5	Right-angle triangle
3	A=2, B=2, C=3	Isosceles triangle
4	A=4, B=4, C=7	Scalene triangle
5	A=1, B=1, C=3	Non-triangle

c) **Boundary Condition for Scalene Triangle:**

Test Case	Condition	Expected Outcome
1	A=2.0, B=3.0, C=5.0	Not a valid triangle
2	A=1.0, B=2.0, C=3.0	Not a valid triangle
3	A=1.0, B=1.0, C=2.0	Not a valid triangle
4	A=0.1, B=0.1, C=0.3	Not a valid triangle
5	A=1.0, B=2.0, C=2.9	Scalene triangle (boundary value)

d) **Boundary Condition for Isosceles Triangle:**

Test Cases	Input Values	Expected Output
1	4 4 7	Isosceles
2	7 4 4	Isosceles
3	4 7 4	Isosceles

e) **Boundary Condition for Equilateral Triangle:**

Test Cases	Input Values	Expected Output
1	1 1 1	Equilateral
2	100 100 100	Equilateral
3	0.1 0.1 0.1	Equilateral

f) Boundary Condition for Right Angle Triangle:

Test Cases	Input Values	Expected Output
1	3 4 5	Right Angle
2	5 3 4	Right Angle
3	4 5 3	Right Angle
4	6 8 10	Right Angle
5	8 10 6	Right Angle
6	10 6 8	Right Angle
7	1 1.4 1.7	Not Right Angle
8	1.4 1 1.7	Not Right Angle
9	1.7 1.4 1	Not Right Angle
10	1.414213 1 2	Right Angle
11	1 1.414213 2	Right Angle
12	1.414213 2 1	Right Angle
13	2 1.414213 1	Right Angle
14	2 1 1.414213	Right Angle
15	1 2 1.414213	Right Angle
16	1.414213 1.414213 2	Not Right Angle

g) For the non-triangle case, identify test cases to explore the boundary.

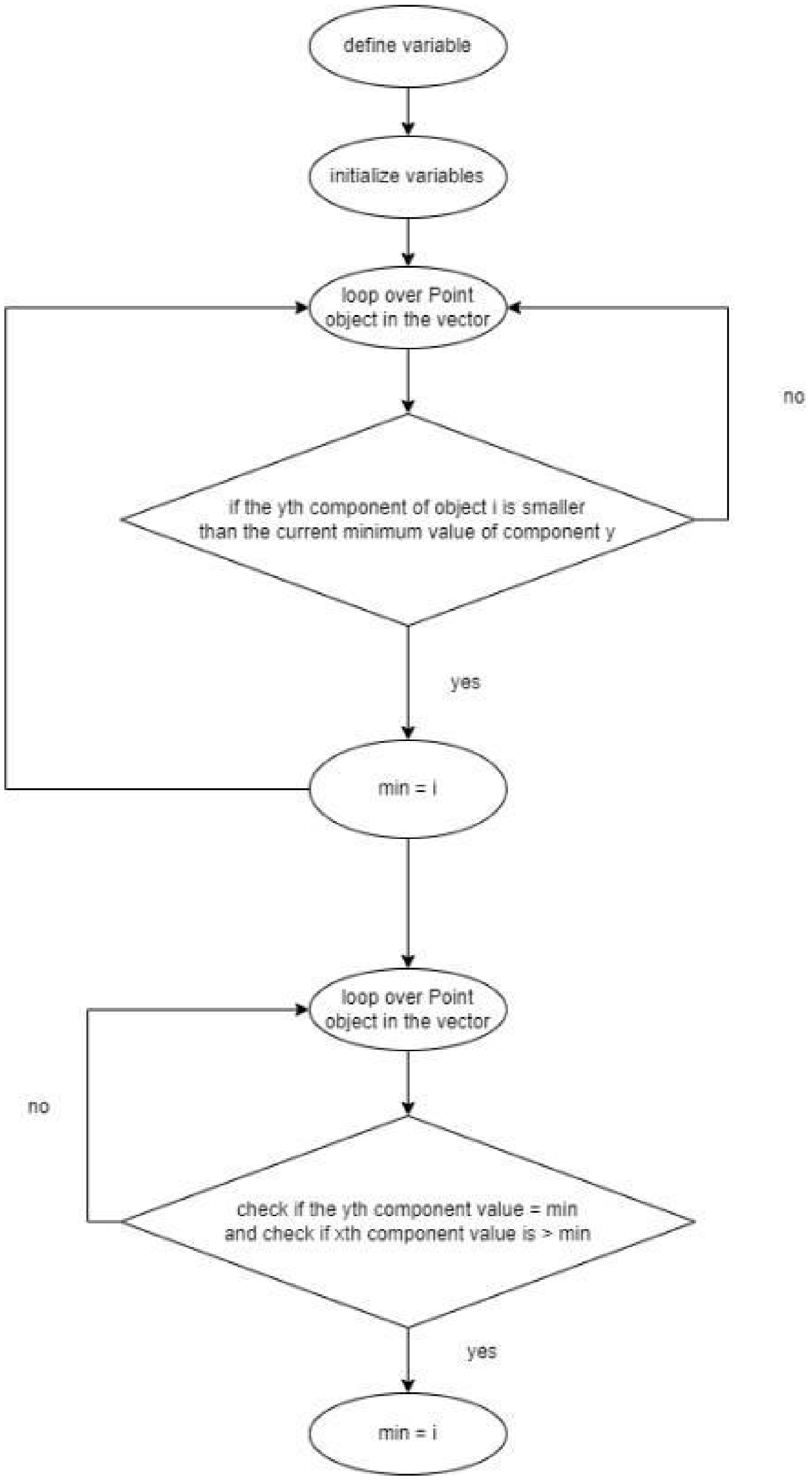
Test Case	Input Values	Expected Output
1	A = 0, B = 1, C = 2	Not a triangle
2	A = 1, B = 0, C = 2	Not a triangle
3	A = 1, B = 2, C = 0	Not a triangle
4	A = 1, B = 2, C = 3	Not a triangle
5	A = 2, B = 3, C = 5	Not a triangle

h) For non-positive input, identify test points.

Test Case	Input Values	Expected Output
1	A = -1, B = 2, C = 3	Invalid input
2	A = 1, B = -2, C = 3	Invalid input
3	A = 1, B = 2, C = -3	Invalid input
4	A = -1, B = -2, C = -3	Invalid input
5	A = 0, B = 1, C = 2	Invalid input

Section B:

1) **Control flow graph**



define variable
initialize variables
loop over Point object in the vector
no
if the yth component of object i is smaller than the current minimum value of component y
yes
min = i
loop over Point object in the vector
no
check if the yth component value = min and check if xth component value is > min
yes
min = i

2)

a. Statement Coverage:

Test Case	Input	Expected Output
1	p = []	Empty vector
2	p = [(1,1)]	Vector with single point
3	p = [(1,1), (2,2)]	Vector with two points
4	p = [(1,1), (2,2), (3,1)]	Vector with three points
5	p = [(1,1), (2,2), (3,1), (4,3)]	Vector with four points

b. Branch Coverage:

Test Case	Input	Expected Output
1	p = []	Empty vector
2	p = [(1,1)]	Vector with single point
3	p = [(1,1), (2,2)]	Vector with two points
4	p = [(1,1), (2,2), (3,1)]	Vector with three points
5	p = [(1,1), (2,2), (3,1), (4,3)]	Vector with four points
6	p = [(1,2), (3,1), (2,1)]	Vector with three points in different order

c. Basic Condition Coverage:

Test Case	Input	Expected Output
1	p = []	Empty vector
2	p = [(1,1)]	Vector with single point
3	p = [(1,1), (2,2)]	Vector with two points
4	p = [(1,1), (2,2), (3,1)]	Vector with three points
5	p = [(1,1), (2,2), (3,1), (4,3)]	Vector with four points
6	p = [(1,2), (3,1), (2,1)]	Vector with three points in different order
7	p = [(1,1), (1,1), (1,1)]	Vector with three identical points
8	p = [(1,1), (2,2), (1,1)]	Vector with two identical points
9	p = [(1,1), (1,2), (2,1)]	Vector with two points with same y component